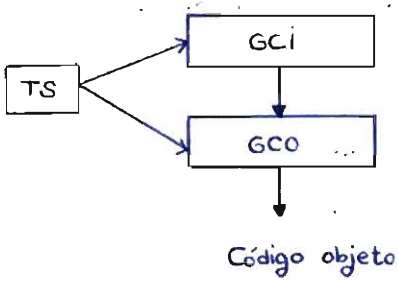
TEMA 9: GENERACIÓN DE CÓDIGO OBJETO (CÓDIGO FINAL).

Este módulo recibe como entrada el programa en código intermedio y la información contenida en la Tabla de Símbolos. Obtiene como salida el programa ejecutable, el código final.



Existen 2 soluciones:

* Generar lenguaje binario.
* Generar lenguaje ensamblador.

Formato de las instrucciones → codigo_op fuente destino

* MODOS DE DIRECCIONAMIENTO:

Absoluto:	M	Dirección de memoria M.
Registro:	R	Contenido del registro R.
Indexado:	C(R)	C + Contenido (R).
Registro indirecto:	*R	Contenido (R).
Indexado indirecto:	*C(R)	Contenido (C + Contenido (R))
Literal (dato inmediato):	#C	Constante C.

1. CÓDIGOS DE MÁQUINA FINAL.

Lo ideal sería obtener código objeto correcto de alta calidad que aproveche al máximo los recursos de la máquina y sea tan eficiente como sea posible.

El código ideal maneja la memoria, selecciona las instrucciones (coge las mejores, no sólo las válidas), asigna los registros de la mejor manera posible y establece el mejor orden para la evaluación de instrucciones.

Sin embargo, hay que ser prácticos y nos conformamos con un código que funcione. Un método sencillo para generar el código objeto es, para cada instrucción de entrada, generar siempre la misma salida:

① x := y + z —GCF→
MOV y, R0 / LD y, R0
ADD z, R0
MOV R0, x / ST R0, x

② x := y * z —GCF→
MOV y, R0
MUL z, R0
MOV R0, z

③ a := a + 1 —GCF→
MOV a, R0
ADD #1, R0
MOV R0, a
} INC a
↳ Mucho más eficiente.

④ goto etiqueta —GCF→ JUMP / BF / GOTO

⑤ if x rel_op y goto etiqueta —GCF→
CMP x y
Bcondición

⑥ PARAM a —GCF→ MOV a, posición_memoria

⑦ CALL suma, 3 —GCF→ Secuencia de llamada.

Vamos a usar una asignación de memoria dinámica mediante pila para los registros de activación. => No hace falta Puntero de Control.

Secuencia de llamada:

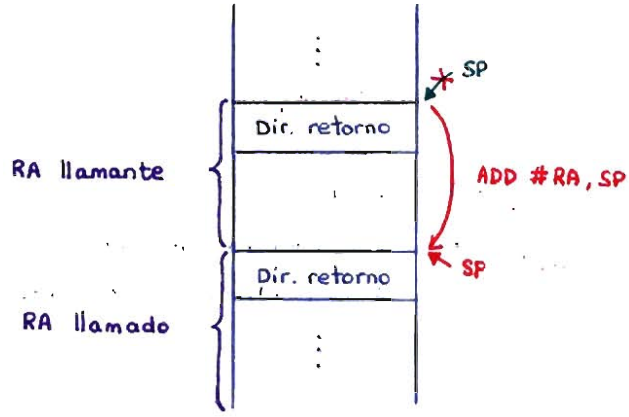
```
ADD   # Tam_RA_llamante , SP      -- Incrementa el puntero de cima de pila.
MOVE  # esta_instrucción + (16), *SP   -- Salva la dirección de retorno.
GOTO  area_codigo_llamado        -- Salta a la primera instrucción del código
                                    del llamado.
```

(16) → Lo que ocupe la instrucción.

Asumimos que la pila crece hacia abajo (hacia direcciones crecientes) y el puntero de la cima de la pila apunta a la dirección de retorno. La dirección de retorno la guardo en la primera palabra del registro de activación.

Para utilizar asignación dinámica mediante pila hay que inicializar el SP en el programa principal:

```
MOVE  # inicio_pila, SP
```



Secuencia de llamada (más completa):

```
ADD   # Tam_RA-llamante , SP      -- Incrementa el puntero de cima de pila.
MOV   # esta_instrucción + 48, (0)SP   -- Salva la dirección de retorno.
MOV   R0, SP - RA_llamante
ADD   #2, R0                     -- R0 = EA del llamante.
MOV   *R0, R0                    -- Repetir prog_A - prog_B + 1 veces.
MOV   R0 , (2)SP                 -- Poner el EA del llamado.
GOTO  etiqueta                   -- Salto a la primera instrucción del código
                                    del llamado.
```

Ejemplo:

Programa fuente: B(b, e+d)

Código intermedio:

```
t1 := e + d
Param b
Param t1
Call B
```

Secuencia de llamada:

- Establecer la dir. retorno en SP + tamaño(RA_A).
- Guardar el estado de la máquina (EM).
- Move "b" a RA_B (SP + tamaño(RA_A) + 2). Si el pasara por dirección sería dir_b
- Move "t_1" a RA_B (SP + tamaño(RA_A) + 2 + tamaño(b)).
- Establecer puntero de acceso (EA).
- Establecer puntero de control (PC). ⇒ No hace falta (no hay datos dinámicos).
- Add: SP + tamaño(RA_A).
- Goto B.

⑧ RETURN —GCi→ Secuencia de retorno.

Hay que devolver el control al llamante.

Secuencia de retorno:

Llamado { GOTO *(SP) -- Salto a la dirección de retorno.

Llamante { SUB # Tam_RA_llamante, SP -- Restaura el SP.
MOV # Tam_RA_llamante + d(SP), V_D -- Almacena el valor devuelto donde esté previsto.

El SP también lo puede hacer el procedimiento llamado antes de devolver el control al llamante, es decir, antes del GOTO.

Si utilizáramos asignación estática de memoria en lugar de una pila, los registros de activación se colocarían en zonas (posiciones) determinadas de memoria. En ese caso sería:

```
Call :  100  MOVE  # esta_instrucción + 20, Área_estática_llamado
        112  GOTO  Primera_instrucción_llamado
        120
Return:      GOTO  * Área_estática_llamado
```

Ejemplo: Asignación de memoria dinámica mediante pila para los registros de activación.

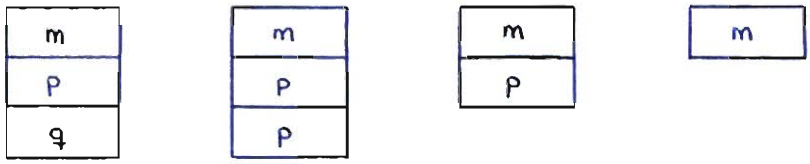
```
Procedimiento m: Acción 1
                 call p
                 Acción 2
                 halt

Procedimiento p: Acción 3
                 call q
                 Acción 4
                 call p
                 Acción 5
                 Return

Procedimiento q: Acción 6
                 Return
```

→

* Registro de activación dinámico utilizando la pila:



* Código final que genera:

```
        ; Código de m
100     MOV # 1000, SP          ; Inicializar la pila.
108     Acción 1                ; Las acciones ocupan más de una instrucción.
148     ADD # TamRAm, SP
156     MOV # 172, * SP         ; Dir. retorno a "m".
164     GOTO 300                ; Dir. comienzo de "p".
172     SUB # TamRAm, SP
180     Acción 2
220     Halt                    ; Fin código "m" (Halt).
        ; Código de p
300     Acción 3
340     ADD # TamRAp, SP
348     MOV # 364, * SP         ; Dir. retorno a "p".
356     GOTO 500                ; Dir. comienzo de "q".
364     SUB # TamRAp, SP
372     Acción 4
412     ADD # TamRAp, SP
420     MOV # 436, * SP         ; Dir. retorno a "p".
428     GOTO 300                ; Dir. comienzo de "p".
436     SUB # TamRAp, SP
444     Acción 5
484     GOTO *(SP)              ; Fin código "p" (Return).
        ; Código de q
500     Acción 6
540     GOTO *(SP)              ; Fin código "q" (Return).
```

+8

Ejemplo: Asignación de memoria estática para los registros de activación.

Procedimiento m: Acción 1
Call p
Acción 2
Halt

Procedimiento p: Acción 3
Return

Asignación de memoria estática ⇒ RA de m: Dirección 300
RA de p: Dirección 400

```
        ; Código de m
100     Acción 1          ; Las acciones están formadas por más de una instrucción.
140     MOV #160, 400     ; Salva dir. retorno de "m" en RAp.
152     GOTO 204          ; Dir. comienzo de "q".
160     Acción 2
200     Halt
        ; Código de p
204     Acción 3
244     GOTO * 400
```

↑ Dirección código (instrucciones)

Dirección del código ≠ Dirección del RA.

1.2. ACCESO A LAS VARIABLES.

Si hay anidamiento tendremos:

1) VARIABLES GLOBALES:

Las variables globales se encuentran en la zona de memoria a continuación del código (dirección estática y conocida). Para calcular la dirección de la variable global es necesario:

- Dirección de comienzo de las variables globales.
- Desplazamiento de la variable (lo tengo en la Tabla de Símbolos).

Dirección de x = Inicio_var_estaticas + despl(TS)

x := 0 —GIO→ MOV #0, (desp)[Inicio_estaticas]

2) VARIABLES LOCALES:

Las variables locales se encuentran en el registro de activación de ese procedimiento (en el RA actual). La dirección de la variable, por tanto, se calcula a partir del SP (dirección relativa al SP).

Dirección de x = SP + Desplazamiento dentro del RA para alcanzar el comienzo de las variables locales + despl(TS)

x := 0 —GCO→ MOV #0, Despl+despl(SP)

3) VARIABLES NO LOCALES:

Las variables no locales se encuentran en sus registros de activación correspondientes. Para calcular la dirección de la variable es necesario:

- Número de punteros de acceso que hay que recorrer (número de saltos).
- Desplazamiento dentro del RA para encontrar el puntero de acceso (PA).
- Desplazamiento de la variable dentro de su RA (TS).

x := 0 —GCO→

```
MOV  SP, R6      ; R6 = Registro temporal dentro del RA.
ADD  #18, R6     ; R6 = PA.
MOV  *R6, R6     } Repetir tantas veces como saltos haya que dar.
ADD  #32, R6
MOV  #0, R6
```

Otra forma:

x := 0 —GCO→

```
MOV  *2(SP), R6   ; R6 = PA
MOV  #0, d'(R6)
```

↳ Sólo un salto: Accede al padre a través del PA y, una vez en el padre, añade su correspondiente desplazamiento.

La dirección de las variables se determina en tiempo de compilación. Esta búsqueda de variables NUNCA se hace en ensamblador. En tiempo de compilación tengo todos los datos necesarios: tamaños, direcciones, desplaz. etc.

2. ALGORITMO DE GENERACIÓN.

Este algoritmo se basa en que conocemos la información contenida en cada registro y en las direcciones en cada instante:

- **Descriptor de registros:** Indica qué hay en los registros en cada instante. Se consulta cada vez que se necesita un registro nuevo. Se asume que inicialmente todos los registros están vacíos.

- **Descriptor de direcciones:** Indica la posición o conjunto de posiciones donde se encuentra el valor de una variable durante la ejecución. Esta posición será un registro y/o una posición de memoria.

Supuestos del algoritmo:

* Para cada operación de código intermedio hay una operación de código final.

* Los resultados de los cálculos serán mantenidos en los registros tanto tiempo como sea posible. Sólo los limpiaremos si el registro se necesita para otro cálculo o si va a haber un salto en el código.

- **Bloque básico (BB):** Conjunto de instrucciones de entrada (código de 3 direcciones) que se ejecutan secuencialmente sin ninguna bifurcación (salto).

UN ALGORITMO PARA GENERACIÓN DE CÓDIGO FINAL

El algoritmo de generación de código objeto que se propone aquí toma como entrada una secuencia de proposiciones de tres direcciones que constituyen un bloque básico (se ejecutan secuencialmente). "instrucciones

- Para cada proposición de tres direcciones de la forma $x := y\ op\ z$ se realizan las siguientes operaciones:

  1. Se invoca la función *obtenreg* (que se describe más adelante) para determinar la posición L donde se debe guardar el resultado de calcular $y\ op\ z$. Generalmente L será un registro, pero también puede ser una posición de memoria.

  2. Se consulta el descriptor de direcciones para determinar y', (una de) la(s) posición(es) en curso de y. Si el valor de y está en ese momento en memoria y en un registro, se prefiere como y' el registro. Si el valor de y no está todavía en L, se genera la instrucción ***MOV y', L*** para colocar una copia de y en L.

  3. Se genera la instrucción ***op z', L*** donde z' es una posición en curso de z. De nuevo, se prefiere un registro a una posición de memoria si z se encuentra en ambos. Se actualiza el descriptor de direcciones para indicar que x está en la posición L. Si L es un registro, se actualiza su descriptor para indicar que contiene el valor de x, y se elimina x de todos los otros descriptores de registros.

  4. Si los valores en curso de y o z, o ambos, no se van a usar más tarde, no están activos a la salida del bloque, y están en registros, se altera el descriptor de registros para indicar que después de la ejecución de $x := y\ op\ z$ dichos registros ya no contendrán y o z, o ambos, respectivamente.

- Una vez que se hayan procesado todas las proposiciones de tres direcciones del bloque básico, mediante instrucciones ***MOV*** se almacenan en memoria aquellos nombres que estén activos a la salida del bloque y que estén sólo en registros. Para ello se necesita consultar: el descriptor de registros, para determinar qué nombres han quedado en los registros; el descriptor de direcciones, para determinar que el mismo nombre no está ya en su posición de memoria; y la información sobre variables activas para determinar si se necesita almacenar el nombre. Si no se ha calculado la información sobre variables activas mediante el análisis del flujo de datos entre los bloques básicos, se debe asumir que todos los nombres definidos por el usuario están activos al final del bloque.

Busca el primer operando (preferiblemente en un registro) y lo lleva a L. Si ya estaba en L no hago nada.

x:= Resultado de la operación.

Puesto que este algoritmo recibe como entrada un solo bloque básico, habría que repetir el mismo proceso para cada uno de los bloques básicos hasta terminar el programa.

NOTAS.

Si la proposición de tres direcciones en curso tiene un operador unario, los pasos son análogos a los descritos, por lo que se omiten los detalles.

Un caso especial importante es el de la proposición de tres direcciones $x := y$.

- ✓ Si y está en un registro, simplemente se cambian los descriptores de registros y de direcciones para consignar que el valor de x ahora se encuentra en el registro que contenía (y aún sigue conteniendo) el valor de y. (Como es lógico, en el caso en que y no se vaya a usar más tarde, y no esté activo a la salida del bloque, ya no interesará seguir manteniéndolo en el registro). (2)

  → El mismo registro contiene a "x" e "y".

- ✓ Si y sólo se encuentra en memoria, en principio se podría hacer constar que el valor de x está en la posición de y, pero esta opción complicaría el algoritmo porque entonces no se podría modificar el valor de y sin modificar el valor de x. Por tanto, si y se encuentra en memoria se utiliza *obtenreg* para encontrar un registro en el que cargar y y convertir ese registro en la ubicación de x.

- ✓ También se puede generar una instrucción ***MOV y, x***, que sería preferible si el valor de x no se fuera a usar más adelante en el bloque. Vale la pena comentar que casi todas, si no todas, las instrucciones de copia pueden eliminarse utilizando un algoritmo de mejora de bloques y de copia y propagación.

(2) Al terminar el bloque, si "x" e "y" no están activas, hay que hacer 2 instrucciones MOV para ese registro: un MOV para "x" y otro MOV para "y". También hay que tener cuidado con la función obtenreg del paso 1 del algoritmo, pues si pisamos el registro machacamos tanto "x" como "y".

La función *obtenreg*

La función *obtenreg* devuelve la posición L en la que guardar el valor de x para la asignación $x := y\ op\ z$. La que aquí se plantea es una función sencilla que se basa en la información sobre qué variables se van a usar más tarde y cuáles no (esta función se puede mejorar de forma que produzca una opción inteligente para L). Los pasos a seguir son:

1. Si el nombre y está en un registro que no contiene el valor de otros nombres (recuérdese que las instrucciones de copia como $x := y$ podrían hacer que un registro guardara el valor de dos o más variables simultáneamente), e y no está activa y no se va a usar después de la ejecución de $x := y\ op\ z$, entonces se devuelve el registro de y para L (es decir, se puede machacar el valor de y sin problemas). Se actualiza el descriptor de direcciones de y para indicar que y ya no se encuentra en L.

2. Si falla (1), se devuelve como L un registro vacío (si es que hay alguno).

3. Si falla (2), y se da el caso de que o bien x se usa más tarde en el bloque, o bien op es un operador que exige un registro (por ejemplo, un operador de índice), entonces hay que:

   - ✓ encontrar un registro ocupado R (un candidato adecuado puede ser un registro cuyo dato sea el que se vaya a utilizar más tarde, o uno cuyo valor también esté en memoria),
   - ✓ almacenar el valor de R en una posición de memoria (mediante ***MOV R,*** **M**), si es que ese valor todavía no está en una posición de memoria apropiada M (si R contiene el valor de varias variables, se debe generar una instrucción MOV para cada variable que haya que almacenar),
   - ✓ actualizar el descriptor de direcciones de M,
   - ✓ y devolver R como la posición L pedida

   (es decir, si no hay ningún registro disponible y, por algún motivo, la operación exige ser realizada en registro, entonces habrá que vaciar alguno de los registros ocupados).

4. Si x no se utiliza en el bloque, o si no se puede encontrar ningún registro ocupado adecuado, se selecciona la posición de memoria de x como L.

Una función *obtenreg* más sofisticada también consideraría los usos posteriores de x y la conmutatividad del operador op al determinar el registro que contendrá al valor de x.

Ejemplo:

```
t := a + b
u := a * c
r := t - u
d := u + r
```

} Bloque básico en código intermedio.

Inicialmente: a, b, c, d en memoria.
Dos registros (R_0 y R_1) vacíos.

PROPOSICIÓN 3-D	CÓDIGO OBJETO	DESCRIPTOR REGISTROS	DESCRIPTOR DIRECCIONES
		R_0 → vacío R_1 → vacío	a, b, c, d
t := a + b	MOV a, R_0 ADD b, R_0	R_0 → t R_1 → vacío	a, b, c, d t → R_0
u := a * c	MOV a, R_1 MUL c, R_1	R_0 → t R_1 → u	a, b, c, d t → R_0 , u → R_1
r := t - u	SUB R_1, (R_0)$_L$ (*)	R_0 → r „ t ∉ R_0 (machaca t) R_1 → u	a, b, c, d (machaca t) r → R_0 , u → R_1 , t ↛ R_0
d := u + r	ADD R_0, (R_1)$_L$	R_0 → r R_1 → d „ u ∉ R_1 (machaca u)	a, b, c, d (machaca u) r → R_0 , d → R_1 , u ↛ R_1
	MOV R_1, d MOV R_0, r		a, b, c, d, r

↑ Llevamos a memoria.

(*) "t" y "u" están en registros. La función obtenreg me dará R_0 ó R para guardar "r". En este caso me dará R_0 ⟹ L = R_0.
En la siguiente instrucción pasa lo mismo y en este otro caso obtenreg me dará R_1 ⟹ L = R_1.

EXAMEN JUNIO 2004.

Se tiene el siguiente fragmento de un programa fuente en un lenguaje en el que todas las variables son enteras y tienen que estar declaradas. En la máquina destino, tanto las direcciones como los valores enteros ocupan 4 bytes.

```
Procedure uno ();
Var a;
    Procedure dos (x);      // x por valor
    Var b;
        Procedure tres (y, ref z);      // y por valor, z por referencia
        Var c;
        Begin       // tres
            c := 9;
            y := c + y;
            z := c + b;      // (***)
        End;
        Procedure cuatro (x);      // x por valor
        Var d;
        Begin       // cuatro
            b := x;
            If x <= 4 Then d := a;
                      Else cuatro (x - 1);
        End;
    Begin       // dos
        b := 7;
        x := x + b;
        cuatro (a);
        tres (x, a);
    End;
Begin       // uno
    a := 5;
    dos (a);
End;
```

Se pide:

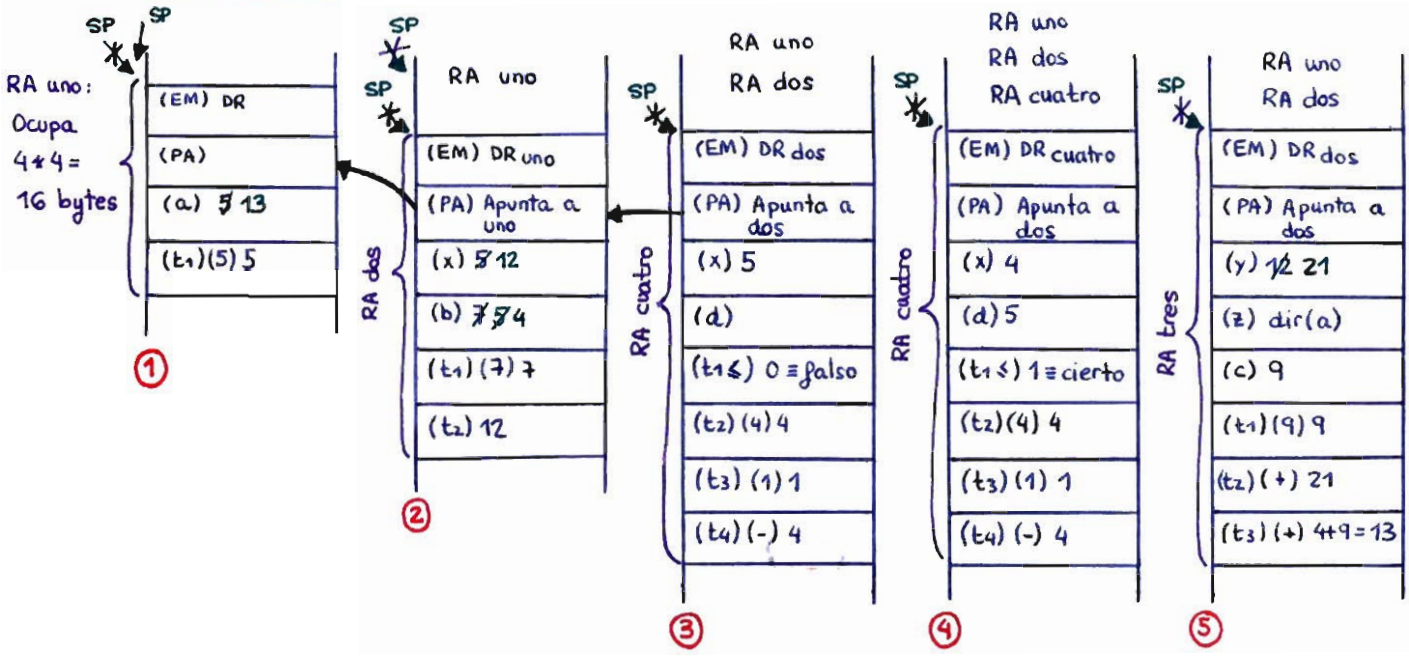
a. Realizar una **traza de ejecución** de este fragmento de programa, dando el diseño del **Registro de Activación** y la pila detallada. Supóngase que todos los temporales van en el Registro de Activación.

Registro de Activación:

Voy a asumir que no hay funciones y que, por tanto, no necesito tener el campo Valor Devuelto. El Puntero de Control tampoco hace falta por estar usando pila.

1.- Estado de la máquina ≈ Dirección de Retorno + Contenido de registros.
2.- Puntero de acceso (PA).
3.- Parámetros.
4.- Variables locales.
5.- Temporales.

Pila detallada:



Dirección de variable local ⇒ Relativa al SP.
Dirección de variable no local ⇒ Relativa al PA.

Ejecución del programa:

① PROCEDIMIENTO uno.

Carga 5 en un temporal (t_1). Luego lo almacena en "a". Ahora aparece la llamada dos(a). Para realizarla pone la Dirección de Retorno (DR) en el Registro de Activación del procedimiento "dos". Establece el Puntero de acceso (PA) al Registro de Activación de "uno" y coloca un 5 en el parámetro "x". Actualiza el SP. (X)

Pasa el control al procedimiento "dos".

② PROCEDIMIENTO dos.

Carga 7 en un temporal (t_1). Luego lo almacena en "b". Hace la suma en el temporal t_2 y el resultado lo guarda en "x". Ahora aparece la llamada cuatro(a). Para realizarla pone la Dirección de Retorno (DR) en el Registro de Activación de "cuatro". Establece el Puntero de acceso al Registro de Activación de "dos" y coloca un 5 en el parámetro "x". Actualiza el SP. (X)

Pasa el control al procedimiento "cuatro".

③ PROCEDIMIENTO cuatro.

Carga el valor de "x" (un 5) en "b". Ahora aparece la comparación. Para realizarla carga 4 en un temporal (t_2) y guarda el resultado en otro temporal (t_1). El resultado es 0 ≡ Falso. Sigue por la rama del else y ahora llega la llamada a "cuatro". El parámetro con el que llama lo almacena en un temporal porque es una resta. Por la llamada pone la Dirección de Retorno (DR) en el Registro de Activación de "cuatro". Establece el Puntero de acceso al Registro de Activación de "dos" y coloca un 4 en el parámetro "x". Actualiza el SP. (X)

Pasa el control al procedimiento "cuatro".

④ PROCEDIMIENTO cuatro.

Carga el valor de "x" (un 4) en "b". Ahora aparece la comparación. Para realizarla carga 4 en un temporal (t_2) y guarda el resultado en otro temporal (t_1). El resultado es 1 ≡ Cierto. Sigue por la rama del then y ejecuta la siguiente instrucción: da a "d" el valor de "a".

El procedimiento "cuatro" acaba. Devuelve el control al procedimiento que le había llamado ("cuatro"). Libera el RA de "cuatro".

⑤ PROCEDIMIENTO cuatro.

Decrementa el SP y se termina el procedimiento. Libera el RA de "cuatro" y devuelve el control al procedimiento que le había llamado ("dos").

⑥ PROCEDIMIENTO dos.

Llega la llamada a "tres". Para realizarla pone la Dirección de Retorno (DR) en el Registro de Activación de "tres". Establece el Puntero de acceso al Registro de Activación de "dos" y coloca el valor 12 ("x") en el parámetro "y" y la dirección de la variable "a" en "z". Actualiza el SP. (X)

Pasa el control al procedimiento "tres".

⑦ PROCEDIMIENTO tres.

Carga un 9 en un temporal y lo mete en "c". Hace la suma de la siguiente instrucción y coloca el resultado en un temporal (t_2). Luego lleva el resultado a "y". Para realizar la siguiente suma coloca el resultado en un temporal (t_3) y lo lleva al Registro de Activación de "uno" (lo sé por la dirección de "a").

El procedimiento "tres" y devuelve el control al que le había llamado ("dos").

Libera el RA de "tres". (X)

⑧ PROCEDIMIENTO dos.

Decrementa el SP y cuando ejecuta se termina el procedimiento. Libera el RA de "dos" y devuelve el control al procedimiento que le había llamado ("uno").

⑨ PROCEDIMIENTO uno.

Decrementa el SP y se termina el procedimiento. Libera el Registro de Activación de "uno".

b. Para la sentencia marcada con (***):

b.1. Detallar el **código objeto** que se generaría, sin usar el nombre simbólico de las variables sino sus direcciones reales.

Instrucción: z := c + b;

c = Variable local.

b = Variable no local.

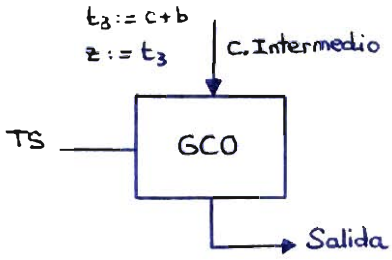
z = Parámetro por referencia.

Código objeto:

```
MOV 16(SP), R1      ; Llevo "c" a R1.
MOV 4(SP), R2       ; R2 contiene la dirección del PA de RA de "dos".
ADD 8(R2), R1       ; Sumo "b" y "c".
MOV R1, 28(SP)      ; Llevo la suma a un temporal (t3).
MOV 28(SP), *12(SP) ; Copio la suma en la dirección almacenada.
```

↳ z es un parámetro por referencia ⇒ Param z = dir de "a".

b.2. ¿Cómo ha podido saber el Generador de Código establecer la dirección para b?



Al compilar, cuando aparece "b" en z := c+b, el analizador léxico se da cuenta de que hay que recorrer una tabla de símbolos:

dif_prof = prof_bloque_uso − prof_bloque_declaración =

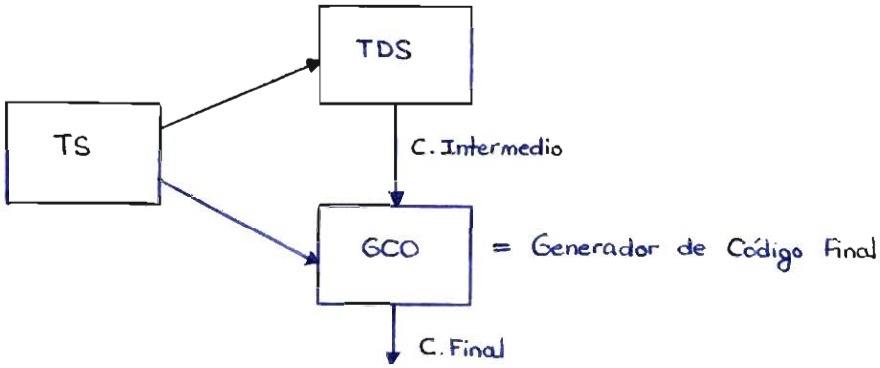
= 3 − 2 = 1

↳ Llego a "b" con una TS o un PA.

b.3. Si la sentencia hubiera sido z := c + d, ¿qué habría pasado? Explicar en detalle, comentando los módulos implicados.

Al llegar a "d", el analizador léxico busca esta variable en la TS, pues como dice el enunciado, las variables deben estar declaradas. Como no está, mira en la TS del padre ("dos"). Como tampoco está, mira en la TS del abuelo ("uno"). Como tampoco está aquí, genera un error ⇒ No genera código ejecutable. Se puede parar la compilación o buscar más errores con el analizador léxico.

c. Escribir detalladamente (no de forma genérica) toda la información que recibe como **entrada** el **Generador de Código Objeto** durante el tiempo en que dicho generador está procesando el procedimiento cuatro, especificando todas y cada una de las entradas de datos al módulo.



La entrada del GCO es el código intermedio correspondiente al procedimiento "cuatro" junto con la Tabla de Símbolos.

Código Intermedio de "cuatro":

```
cuatro :  b.lugar := x.lugar                    b.lugar → 1PA + desplazamiento_b
          t1.lugar := 4
          if x.lugar > t1.lugar goto else
          d.lugar := a.lugar                    a.lugar → 2PA + desplazamiento_a
          goto fin
else :    t2.lugar := 1
          t3.lugar := x.lugar - t2.lugar
          param t3.lugar
          call cuatro
fin :     return  ≈ goto a la instrucción correspondiente.
```

Tablas de símbolos:

TS uno Tam RA : Prof : 1

LEXEMA	TIPO	LUGAR	N_PARAM	PASO_PARAM	TAM_RA	PROF.	PUNT_TS
a	var	0					
dos	n-proc		1	valor	X	2	TS dos →

↓ Lugar = Desplazamiento

TS dos TamRA: X Prof: 2

LEXEMA	TIPO	LUGAR	N_PARAM	PASO_PARAM	TAM_RA	PROF.	PUNT_TS	ETIQ_INICIO
x	p-val	0						
b	var	4						
tres	n-proc		2	valor, ref.	Y	3	TS tres →	
cuatro	n-proc		1	valor	Z	3	TS cuatro →	cuatro

TS cuatro TamRA: Z Prof: 3

LEXEMA	TIPO	LUGAR	N_PARAM	PASO_PARAM	TAM_RA	PROF.	PUNT_TS
x	p-val	0					
d	var	4					
t_1	var	8					
t_2	var	12					
t_3	var	16					

Comentarios:

* En el campo TIPO voy a indicar si es una variable o un parámetro. Indicar que las variables son enteras no tiene sentido porque todas lo son.

* Necesito indicar si es un parámetro por valor o por referencia. Para ello hay dos posibles soluciones:
  - Un campo nuevo en el que indico si el parámetro se pasa por valor o por referencia.
  - Utilizar el campo TIPO poniendo param_valor o param_referencia.

* Valor de desplazamiento:

Lo que he utilizado →
  - El desplazamiento empieza en 0 y es común para los parámetros, las variables y los temporales. Esto me obliga a que en tiempo de ejecución tenga los parámetros, las variables locales y los temporales todos juntos en el registro de activación.
  - Tener dos desplazamientos independientes: uno para los parámetros y otro para variables locales y temporales.

* El tamaño del RA y la profundidad los puedo guardar en la cabecera de la TS correspondiente o bien en la entrada correspondiente de la TS padre. También puedo guardarlo en los dos sitios.

* Campo Etiq_Inicio: Indica la etiqueta de la primera instrucción del código final que voy a generar.

EXAMEN FINAL FEBRERO 2005.

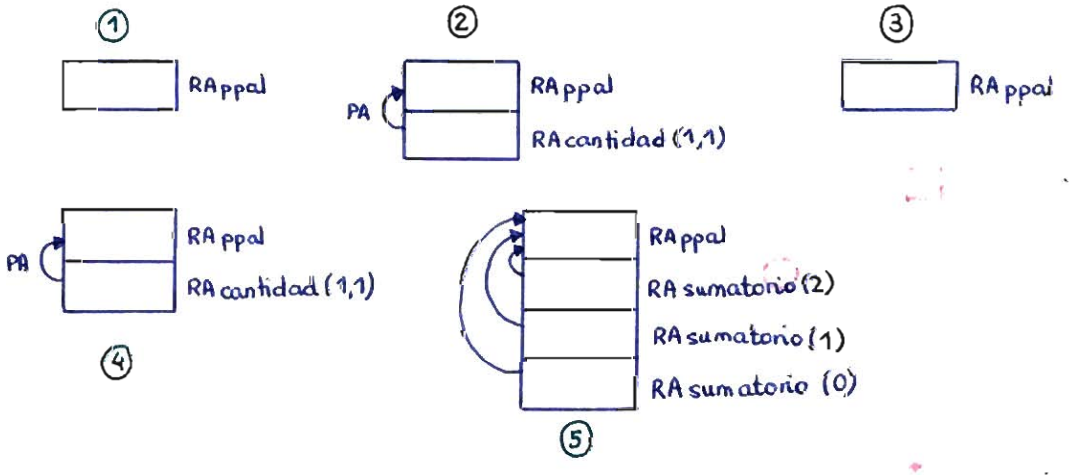
Se tiene un lenguaje que dispone de anidamiento de funciones y procedimientos y en el que el paso de parámetros es siempre por valor. Se pide dibujar, con el mayor detalle posible, la **traza de ejecución** del programa siguiente.

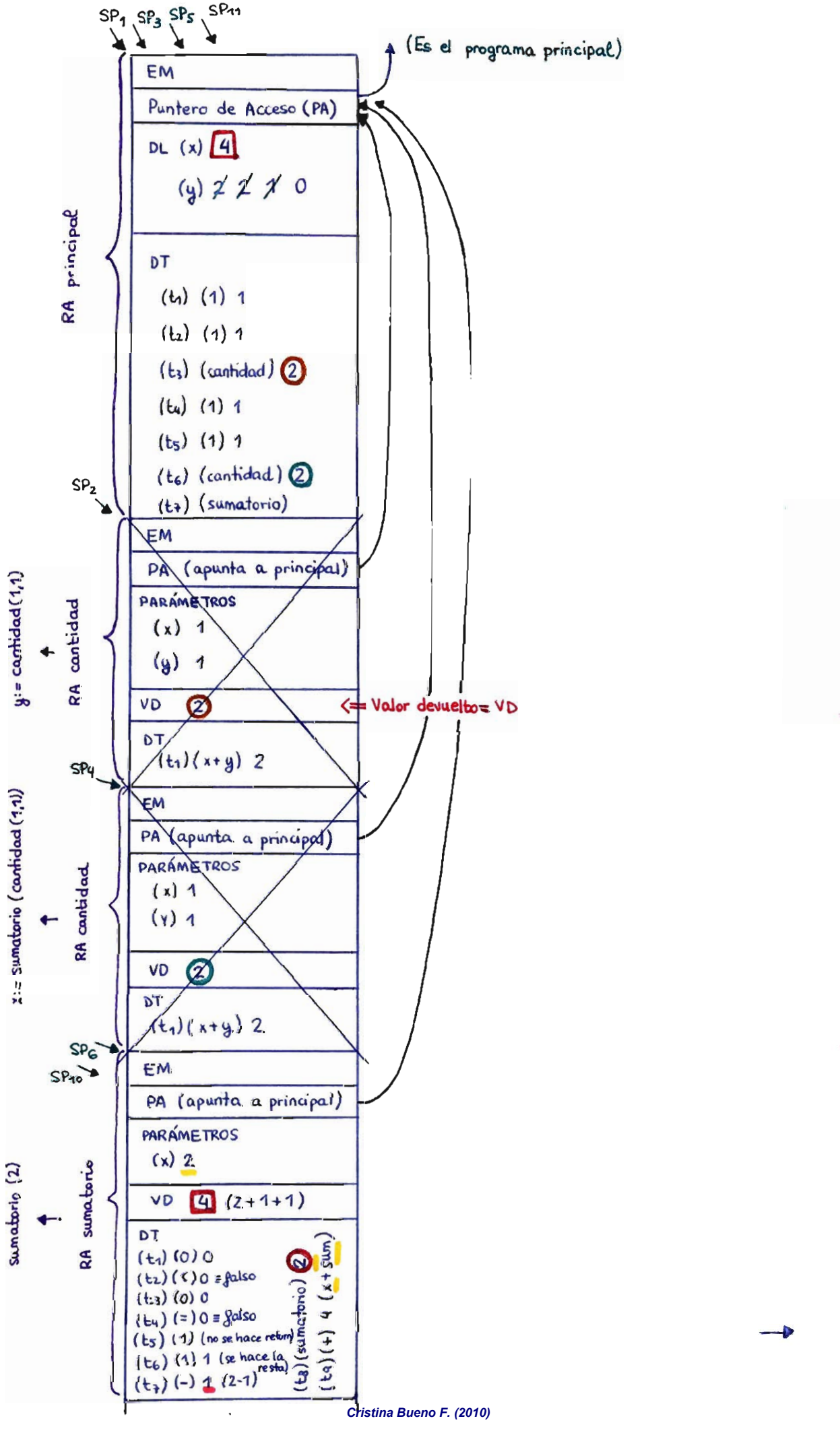
```
Program principal;
Var x, y: integer;
    Function cantidad (x, y: integer): integer;
    begin
          return x + y
    end;
    Function sumatorio (x: integer): Integer;
    begin
          y:= x;
          if (x<0) then write ("El parámetro no debe ser negativo")
            else if (x=0) then return 1
                  else return x + sumatorio (x-1)
    end;
begin       (*principal*)
      y:= cantidad (1,1);
      x:= sumatorio (cantidad (1,1));
      write ("El resultado es ", x)
end       (*principal*).
```

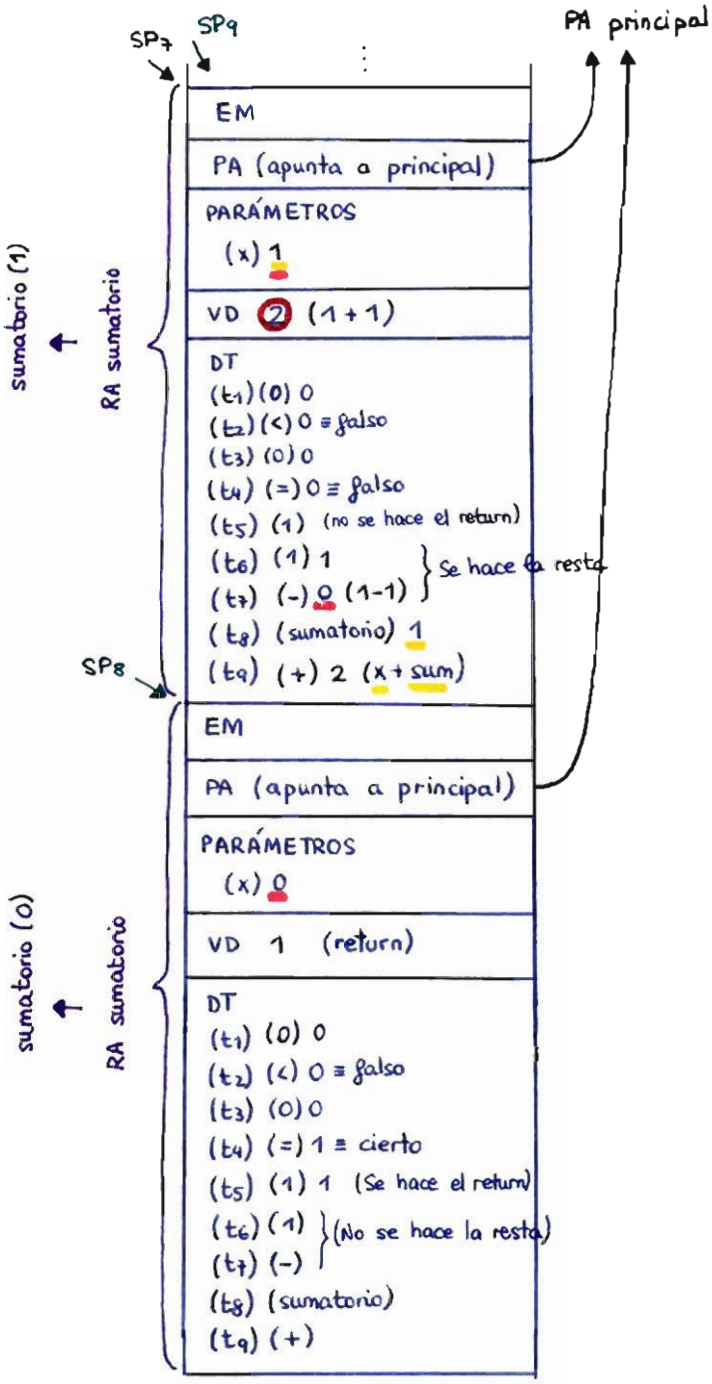
* Primero hay que diseñar los registros de activación:

Estado de la máquina
Puntero de acceso
Parámetros
Datos locales
Valor devuelto
Datos temporales

* Esquema de la traza de ejecución sin desarrollar:




SP1 SP3 SP5 SP11
(Es el programa principal)
EM
Puntero de Acceso (PA)
DL (x) 4
(y) 2 2 1 0
RA principal
DT
(t1) (1) 1
(t2) (1) 1
(t3) (cantidad) 2
(t4) (1) 1
(t5) (1) 1
(t6) (cantidad) 2
(t7) (sumatorio)
SP2
EM
PA (apunta a principal)
PARÁMETROS
(x) 1
(y) 1
VD 2
<== Valor devuelto= VD
y:= cantidad (1,1)
RA cantidad
DT
(t1) (x+y) 2
SP4
EM
PA (apunta a principal)
PARÁMETROS
(x) 1
(y) 1
VD 2
DT
(t1) (x+y) 2
x:= sumatorio (cantidad (1,1))
RA cantidad
SP6
SP10
EM
PA (apunta a principal)
PARÁMETROS
(x) 2
VD 4 (2+1+1)
DT
(t1) (0) 0
(t2) (<) 0 ≡ falso
(t3) (0) 0
(t4) (=) 0 ≡ falso
(t5) (1) (no se hace return)
(t6) (1) 1 (se hace la resta)
(t7) (-) 1 (2-1)
(t8) (sumatorio) 2
(t9) (+) 4 (x+sum)
Sumatorio (2)
RA sumatorio


SP7
SP9
PA principal
EM
PA (apunta a principal)
PARÁMETROS
(x) 1
VD 2 (1+1)
DT
(t1) (0) 0
(t2) (<) 0 ≡ falso
(t3) (0) 0
(t4) (=) 0 ≡ falso
(t5) (1) (no se hace el return)
(t6) (1) 1
(t7) (-) 0 (1-1)
Se hace la resta
(t8) (sumatorio) 1
(t9) (+) 2 (x+sum)
sumatorio (1)
RA sumatorio
SP8
EM
PA (apunta a principal)
PARÁMETROS
(x) 0
VD 1 (return)
DT
(t1) (0) 0
(t2) (<) 0 ≡ falso
(t3) (0) 0
(t4) (=) 1 ≡ cierto
(t5) (1) 1 (Se hace el return)
(t6) (1)
(t7) (-)
(No se hace la resta)
(t8) (sumatorio)
(t9) (+)
sumatorio (0)
RA sumatorio